FACULTY OF INFORMATION TECHNOLOGY CTU IN PRAGUE

ASSIGNMENT OF BACHELOR'S THESIS

Title:	Texture extraction from photographs
Student:	Danil Luzin
Supervisor:	Ing. David Sedláček, Ph.D.
Study Programme:	Informatics
Study Branch:	Web and Software Engineering
Department:	Department of Software Engineering
Validity:	Until the end of summer semester 2018/19

Instructions

1) Design and implement a tool for extracting texture information for a 3D object from photographs with known camera positions and orientations.
2) Research the challenges associated with texture extraction from photographs, study and analyze techniques and approaches used in this field. Design and implement a tool that will for a given 3D object (or set of 3D objects), set of photographs and a camera position data, be able to extract and generate texture for that 3D object.
3) Test tool functionality on datasets provided by the thesis supervisor.

The command line tool (no UI required) should be easily integrated into Bundler SFM/Visual SFM software pipeline and also in laser 3D scanning pipeline.

References

1) Jan Kirschner, Rekonstrukce 3D modelu z fotografií, dip. prac. 2008, ČVUT FEL
2) Petr Beránek, Systém pro automatické snímání pomocí 3D laser scanneru a fotoaparátu, dip. prac. 2017, ČVUT, FEL.
3) Let There Be Color! – Large-Scale Texturing of 3D Reconstructions. Michael Waechter, Nils Moehrle, and Michael Goesele. In: European Conference on Computer Vision (ECCV 2014), Zürich, Switzerland, 6-12 Sept. 2014.
4) Bundler: http://www.cs.cornell.edu/ snavely/bundler/

Ing. Michal Valenta, Ph.D.
Head of Department

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Dean

Prague December 13, 2017

Acknowledgements

I would like to thank my supervisor Ing. David Sedláček, Ph.D. for the help and advice during the development and writing of this thesis. I would also, like to thank my family for always being there to support me.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Article 46(6) of the Act, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the "Work"), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work in any way (including for-profit purposes) that does not detract from its value. This authorization is not limited in terms of time, location and quantity. However, all persons that makes use of the above license shall be obliged to grant a license at least in the same scope as defined above with respect to each and every work that is created (wholly or in part) based on the Work, by modifying the Work, by combining the Work with another work, by including the Work in a collection of works or by adapting the Work (including translation), and at the same time make available the source code of such work at least in a way and scope that are comparable to the way and scope in which the source code of the Work is made available.

In Prague on May 9, 2018 .....

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Luzin, Danil. *Texture Extraction from Photographs.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2018.

Abstract

This bachelor's thesis describes the process of analysis, design and implementation of a texture extraction tool.

Provided with a reconstructed 3D model, set of photographs and a camera calibration data, the tool is able to generate texture file/files for that 3D model. The work describes techniques, approaches and challenges present in the field of texture extraction. Couple new ways of solving some of those challenges are also described and implemented in the work. The result of the work is a fully functional tool written in C++.

Keywords texture extraction, image processing, 3D reconstruction, Bundler, Structure from motion, Markov Random Field, photogrammetry, C++

Abstrakt

Tato bakalářská práce popisuje proces analýzy, návrhu a implementace nástroje pro extrakci textur.

Nástroj vytvoří pro 3D model získaný fotorekonstrukcí obrazové soubory, které se dají použít pro otexturování tohoto 3D modelu. Vstupem programu je 3D model, sada fotografií a kalibrační data kamer. Práce popisuje techniky, přístupy a výzvy přítomné v oboru extrakce textur. V této práci je implementováno pár nových způsobu řešení některých z těchto problému. Výsledkem práce je plně funkční nástroj napsaný v C++.

Klíčová slova extrakce textur, zpracování obrazu, 3D rekonstrukce, Bundler, Structure from motion, Markov Random Field, fotogrammetrie, C++

Contents

List of Figures

List of Tables

Introduction

3D reconstruction has been a subject of research and innovation for well over a decade now. Modern photogrammetry software allows anyone with a decent camera to create highly detailed 3D models from just a couple of photographs, and more advanced users can utilize specialized laser scanners that can produce significantly more accurate reconstructions.

This technology has found numerous applications in several different fields such as entertainment, engineering, historic preservation and education. Number of modern games and movies use 3D scans of real-life objects to add details and realism to their products [1, 2]. Despite that, preservation of color information for those reconstructed models still presents a challenge.

In the recent several years this problem has been researched extensively and several interesting and innovative approaches of solving it have been suggested. Each approach has its limitations and advantages, that are going to be discussed in the research part of this thesis. One of those approaches involves calculating the color for every vertex separately and interpolating it across the polygon. Encoding color information on a per-vertex basis, however, makes it impossible to maintain realism and keep the size of reconstructed models manageable at the same time. That is why, in computer graphics, it is very common to simplify models to reduce number of polygons, and use texture images to store color information for every polygon. But the process of creating high quality textures from images can be very difficult due to inaccuracies in model reconstruction, changes in camera color balance and exposure, as well as sheer size of reconstructed models and number of source images.

The goal of this thesis is to analyze approaches and techniques used in the field of texture extraction as well as to design and implement a tool capable of generating 2D texture images for reconstructed 3D models.

Thesis Structure

This thesis is divided into the following chapters.

Goals and Requirements. Functional and non-functional requirements of the tool are specified here.

Research. In this chapter, current state of texture extraction field is discussed. Several ways this problem has been addressed before are described and compared. The approach, used for the final tool, is picked and discussed in further details.

Analysis and Design. This chapter focuses on the design of the tool. Texture extraction pipeline that was implemented in the tool is outlined here. A couple new ways of solving some minor challenges, involved in the texture extraction process, are described. Different general computer graphics techniques, used in the tool, are also detailed here.

Technologies. The description of the Bundler pipeline and the interpretation of it's output are provided in this chapter. This chapter also describes different tools and libraries that were used during the development.

Implementation. Description of the implementation is detailed in this section of the thesis.

Testing. This chapter is concerned with testing of the tool by providing it with different input data and changes to the configuration. Different parameters and their impact on performance and texture quality is discussed here.

Future Improvement. Ideas on further improvement of the tool and texture extraction pipeline are detailed in this chapter.

Chapter 1

Goals and Requirements

The goal of the research and analysis part of this thesis is to get familiar with state of the art of texture extraction process. After comparing several different approaches, one of them will be picked to be implemented in the tool. That approach will then be studied and described more thoroughly. Major subproblems should be identified in this part of the work and solution for those must be selected or suggested. This research should act as a groundwork for the practical part of the thesis.

The goal of the practical part of the thesis is to implement a tool that would be capable of generating 2D texture images for reconstructed 3D models.

1.1 Requirements

The following list of requirements should act as a precise definition of the functionality and form expected from the tool.

(FURPS[1] labeling convention is used.)

1.1.1 Functional

F1: Texture extraction. Provided with a reconstructed 3D model, set of photographs that were used in the reconstruction process and a camera calibration information associated with those photographs, the tool should be capable of generating texture file/files for that model.

F2: Bundler and Visual SFM compatibility. The tool should be able to use output from the Bundler SFM [4] and Visual SFM [5] for camera calibration and positions.

[1] Functionality, Usability, Reliability, Performance, Supportability. An acronym for the system that provides a framework for classifying architectural requirements [3].

F3: Laser scan compatibility. The tool should be able to use output from the laser 3D scanning software developed in the CTU FEE Department of Computer Graphics and Interaction [6].

F4: Model file format. The tool should accept the Wavefront OBJ[2] files for the model input. Models must be triangulated.

F5: Multiple objects per model The tool should support models comprised of multiple separate objects, and generate separate textures for them.

F6: Configuration validation. The tool should validate configuration before every execution.

F7: Independent processing steps. User must be able to make changes to the configuration of the tool without having to rerun particular time-consuming parts of the extraction pipeline, i.e have an ability to save intermediate calculation results and have an ability to continue the extraction process after changing the configuration without having to do those calculations again.

1.1.2 Non-functional

S1: Portability. C++ code or external libraries used in the tool, must not be operating system specific.

[2]The Wavefront OBJ format is a standard for representing polygonal data in ASCII form. [7]

Chapter 2

Research

The problem seems to be simple. We have a 3D model reconstructed by a specialized software either through photogrammerty or laser scanning, we have photographs depicting the model and we know camera calibration and positions. In the hypothetical ideal scenario reconstruction is absolutely accurate, all photographs are perfect and don't have changes in illumination, exposure, quality or color balance and there is no scale difference between close-ups and distant overviews [8]. Under those assumptions problem indeed has a simple solution: each point on the surface of the model can be back-projected to one of the views to retrieve color at that point. Unsurprisingly in this ideal situation this approach would produce a perfect seamless textures.

In reality, however, all of the aforementioned problems are present. Reconstructed models are only approximated, camera position, orientation and calibration are rarely registered perfectly, source images can have unregistered occludes (e.g. pedestrians for a city building reconstructions) [9]. This obviously complicates the problem quite a bit.

In this chapter different techniques of preserving color information and in particular texture extraction will be discussed and compared.

2.1 Vertex color vs. texture mapping

When it comes to adding color to a 3D model broadly speaking there are two different approaches:

- Encoding color information for every vertex of the model separately.
- Using texture images and texture mapping.

When the first approach is used, the color of every pixel of each polygon is interpolated using the color of the vertices that make up that polygon. This method has its obvious limitations. The level of detail of the model

is restricted by the number of vertices it has. This leads to a problem where models of real objects, even very simple ones, must have thousands or millions of vertices defining them if we want to keep the original look of that object. Consequentially the simplification of the mesh (i.e the reduction of the number of polygons) leads to a loss of detail.

On the other hand, when using texture mapping, a 2D image is applied to a 3D polygon based on a UV coordinates [3] of the vertices that make up that polygon. This approach is ideal for defining high frequency detail often present in the real-life objects. The other benefit of using texture mapping is that the amount of color details is only limited by the resolution of the texture images used.

Since the focus of this thesis is texture extraction for real-life reconstructions, the texture mapping approach is picked.

Image 2.1 show a comparison of two approaches. Difference in quality for the texture approach is quite significant.

Figure 2.1: Comparison of the vertex color and texture approach, using the same model. *Left*: Model with wire frame overlay. *Middle*: Render using the vertex color. *Right*: Render using the texture image.

2.2 Texture extraction

Texture extraction can be defined as a process that takes images that depict a target model as an input and produce texture image for that model as an output.

Currently two major different ways of tackling it can be identified:

- Per texel[4] optimization.
- View assignment and blending.

[3] Coordinates in texture space, assigned as vertex attributes and used for texture lookup.
[4] Texture pixel.

2.2.1 Texel based optimization

This technique works with every texel separately.

For every pixel in a texture map we can produce a sample list, where samples will represent the color at the same point on the surface of the 3D model, but taken from the different source images (obviously excluding the images where that point in not visible). After such list is created each pixel could have multiple color candidates. Different implementations of texel based approach deal with calculating the final color based on those candidates differently.

Easiest way of dealing with it would be just to average out all of the candidates to get the final color [10]. This, however, produces noticeably blurry textures, and reconstruction inaccuracies can cause ghosting effect in places of view-model misalignment [11].

Slightly more sophisticated method can take into a account source image quality and pick only one candidate. For example, a candidate can be picked if the view associated with it shows the point at the rightmost angle, or the point appears to be the closest to the camera among the candidates. Unfortunately this approach also has its shortfalls. Simple heuristic that determines one right candidate is often not enough. It doesn't suffer from blurriness that much, but the issue of color inconsistency is amplified by the lighting differences between input images.

More complicated methods construct one or several weight maps for each view and then use them to calculate the final color [12]. This approach can be very flexible. Many different weight criteria can be defined and the final color is calculated as a weighted average of many color candidates. These are some examples of common weight criteria types:

- Angle. Camera orthogonality to the corresponding sample point can play a role in affecting the weight. Samples taken from the polygons at extreme angles can be discarded or made less suitable by the weight map.

- Depth. Samples that are closer will affect the color more than those that were take from the distance.

- Overlap potential. If one part of the model overlaps with another from a certain point of view, we can define an area of pixels around the overlap that will play a lesser role in the construction of the final color [13]. This helps with cases where reconstruction edges are not precise, something that is common in practice.

2.2.2 View assignment

Opposite to the per texel approach, this method operates on each polygon as a whole. Generally only one view (photograph) is picked to texture a

whole polygon. After that, the generated texture patches can be optimized in several different ways to minimize the seams between texture patches taken from different source images. Various implementations of this method differ in how they determine which view to use for which polygon, or how texture patches are later modified.

Simpler form of this technique would work similarly to picking one single color candidate for every texel. A source image can be picked to texture a polygon based on how good the texture patch is. One can come up with many different criteria of determining the quality of a source image, but in the end this technique alone is not enough. Its main flaw lays in the fact that every polygon is viewed in isolation, this leads to very noticeable color discrepancies (seams) in areas where two different views were picked to texture the adjacent polygons.

In order to minimize the seams caused by the reconstruction imprecision Eisemann et al. [14] suggest to warp the projected texture patches. And another approach suggested recently by Jeon et al. [15] picks a couple of key frames that are later used to get a final texture patch by optimizing and warping single key frames and then blending them together for every polygon.

However, a more flexible way of alleviate the difficulty of view selection and seam minimization was proposed by Lempitsky and Ivanov [16]. They pose this problem as a pairwise Markov Random Field (MRF) problem. Since their work and its derivatives will form the base of our implementation, it will be discussed in more details in the next section 2.3.

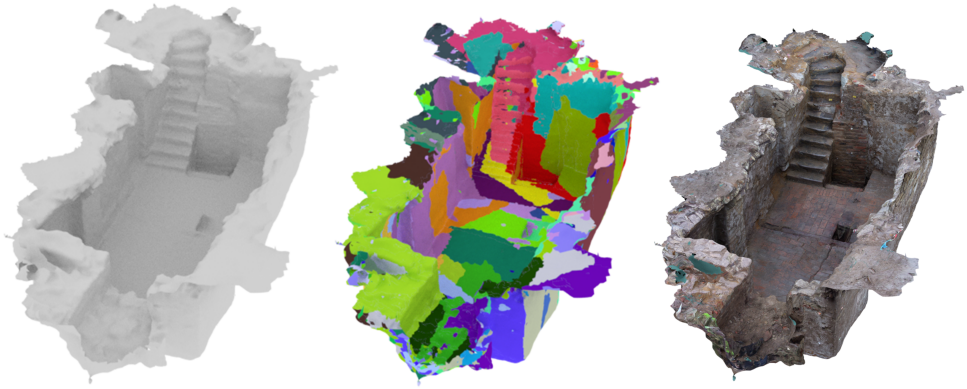

Figure 2.2: Simplified labeling visualization. *Left:* input model. *Middle:* labeling. Each color signifies different color being picked for the polygon. *Right:* final render using the generated texture.

2.3 MRF approach

The overall texture quality depends on two aspects:

1. Quality of a polygon texture patch in and of itself.
2. Quality of a seam created by the stitching of separate texture patches together.

Therefore, we can define the process of texture extraction as an image stitching problem. The goal is to pick the best texture source and to penalize mismatches across polygon boundaries. [9]. This results in a Markov Random Field problem and, as mention before, this technique was suggested by Lempitsky and Ivanov [16].

But before the approach can be discussed further, first a very brief understanding of what the MRF is and what its solution entails is required.

2.3.1 MRF in computer vision

"A variety of computer vision problems can be optimally posed as Bayesian labeling in which the solution of a problem is defined as the maximum posteriori (MAP) probability estimate of the true labeling . . . the best one we can get from random observations" [17, page 1]

Markov Random Field theory is in its core a probabilistic tool that has its roots in physics research of ferromagnetic materials [18]. Its main strong suite is that is allows to encode and enforce prior contextual constraints in a quantitative way.

In the field of computer vision the reason the MRF model is used is because it allows to define a systematic approach to solving certain problems:

1. Represent the problem as one of the labeling in which optimal solution is defined as MAP label configuration (i.e most probable, given the prior context).
2. Define the likelihood and contextual constraints.
3. Compute the MAP labeling by minimizing a posterior energy. [17]

In order to better understand the process stated above, it must be defined more formally. The following equations and definitions are based on [19, 17].

A labeling problem can be specified with a set of targets and a set of labels. Let $\mathbf{t}$ be a set of m discrete targets

$$t = \{1, \ldots, m\} \tag{2.1}$$

Let $\mathbf{L}$ be a set of M labels.

$$L = \{1, \ldots, M\} \tag{2.2}$$

Labeling, therefore, is a process of assign a label from $\mathbf{L}$ to each of targets in $\mathbf{t}$. For our purposes only the discrete labeling is allowed, which means that the value of label assigned to a target i can be defined as

$$f_i \in \mathbf{L} \tag{2.3}$$

Let $F = \{F_1, \ldots F_m\}$ be a family of random variables defined on t, where each random variable F_i takes a value in $\mathbf{L}$. A joint event $\{F_1 = f_1, \ldots, F_m = f_m\}$ abbreviated $F = f$, is a realization of F where $f = \{f_1, \ldots, f_m\}$ is called a configuration of F.

A configuration f can be interpreted as a mapping $f : \mathbf{t} \to \mathbf{L}$, or as a labeling $\{f_1, \ldots, f_m\}$ of the targets.

The set of all configurations is

$$\mathbf{S} = \mathbf{L}^m = \underbrace{\mathbf{L} \times \mathbf{L} \cdots \times \mathbf{L}}_{m \text{ times}} \tag{2.4}$$

$P(f \mid r)$ measures the probability of a labeling, given the observation $\mathbf{r}$. This means that our goal is to find an optimal labeling $\hat{f}$ which maximizes $P(f \mid \mathbf{r})$.

$$\hat{f} = \arg\max_{f \in \mathbf{S}} P(f \mid \mathbf{r}) \tag{2.5}$$

To calculate posterior probability a Bayesian rule can be used:

$$P(f \mid \mathbf{r}) = \frac{p(\mathbf{r} \mid f)P(f)}{p(\mathbf{r})} \tag{2.6}$$

Since $p(\mathbf{r})$, the density function of $\mathbf{r}$, is constant it does not affect the MAP solution. So in order to find a MAP solution we have to define prior probabilities $P(f)$ and a likelihood function $P(\mathbf{r} \mid f)$.

To help with the definition of those, a Hammsley-Clifford theorem [20] can be utilized. To use it, a slightly different outlook on the MRF model must first be defined. Namely a graph representation of the MRF.

Let $\mathcal{N}$ be a neighborhood system of the targets t

$$\mathcal{N} = \{\mathcal{N}_i \mid \forall i \in t\} \tag{2.7}$$

where $\mathcal{N}_i$ is a set of neighboring targets to i for which $i \notin \mathcal{N}_i$ and $i \in \mathcal{N}_q \Leftrightarrow q \in \mathcal{N}_i$. Based on this definition the pair $(t, \mathcal{N})$ is a graph in the usual sense.

Further a clique of this graph is defined traditionally as a subset $c \subseteq t$ where every pair of targets in this subset are neighbors. Clique of the n^{th} order contains n targets. We can denote the collections of all cliques of first order, all cliques of second order, ..., all cliques of m^{th} order, by $C_1, C_2, \ldots, C_m$ respectively. Set of all cliques in a $(t, \mathcal{N})$ graph is denoted by C.

$$C = C_1 \cup C_2 \cup \cdots \cup C_m \tag{2.8}$$

According to Hammsley-Clifford theorem [20] F is an MRF if and only if probability distribution $P(F = f)$ follows a Gibbs distribution. A simplified version is presented:

$$P(f) = \frac{1}{Z} \exp(-U(f)) \tag{2.9}$$

Z is a normalization constant and $U(f)$ is a prior energy. Prior energy is then defined as follows:

$$U(f) = \sum_{c \in C} V_c(f) = \sum_{\{i\} \in C_1} V_{C_1}(f_i) + \sum_{\{i,j\} \in C_2} V_{C_2}(f_i, f_j) + \ldots \tag{2.10}$$

V_c is something that is called a clique potential. By defining those potentials $V_c(f)$ for all $c \in C$ we can precisely define *a priori* knowledge of interactions between labels assigned to neighboring targets, and how individual labels affect each other.

The last piece of the puzzle is a posterior energy formulation. If we express likelihood in the exponential form

$$p(\mathbf{r} \mid f) = \frac{1}{Z_{\mathbf{r}}} \exp(-U(\mathbf{r} \mid f)) \tag{2.11}$$

Where $Z_{\mathbf{r}}$ is again a normalization constant, and $U(\mathbf{r} \mid f)$ is called the likelihood energy. Posterior probability is then

$$p(f \mid \mathbf{r}) = \frac{1}{Z_E} \exp(-E(f)) \tag{2.12}$$

and finally, posterior energy is

$$E(f) = U(f \mid \mathbf{r}) = U(f) + U(\mathbf{r} \mid f) \tag{2.13}$$

The MAP solution is then equivalently can be found by

$$\hat{f} = \arg\max_{f \in \mathbf{S}} P(f \mid \mathbf{r}) = \arg\min_{f \in \mathbf{S}} E(f) \tag{2.14}$$

To summarize, the MRF modeling process can be broken down into the following steps:

1. Defining of the $\mathcal{N}$ neighboring system.
2. Defining of cliques C.
3. Defining prior clique potential V_c.
4. Deriving a likelihood energy $U(\mathbf{r} \mid f)$.
5. Deriving the posterior energy $E(f)$.
6. Getting of the MAP labeling by minimizing the posterior energy $E(f)$. [17]

2.3.2 MRF modeling for texture extraction

In order to apply the modeling framework outlined in the section 2.3.1 first we specify the neighboring system $\mathcal{N}$ as one that mirrors the topological relations of the polygons in the mesh.

In our model targets t are the polygons $t = \{p_1, p_2, \ldots, p_m\}$, and the neighboring system $\mathcal{N}$ is defined by the polygon adjacency:

$$(p_i \in \mathcal{N}_{p_q}) \Leftrightarrow (p_q \in \mathcal{N}_{p_i}) \Leftrightarrow (p_i \text{ and } p_q \text{ share the same edge}) \tag{2.15}$$

We then define $\mathbf{L}$ as a set of source images $L = \{v_1, v_2, \ldots, v_M\}$. Therefore, the labeling f can be viewed as an assignment of source images to the polygons to be used for texturing.

To adapt the MRF framework for the texture extraction Lempitsky and Ivanov [16] suggest the following variation of the energy term 2.13 (provided simplified version is based on [8]):

$$E(f) = \sum_{p_i \in t} E_{\text{data}}(p_i, f_i) + \sum_{(p_i, p_q) \in C_2} E_{\text{smooth}}(p_i, p_q, f_i, f_q) \tag{2.16}$$

The data term E_{data} prefers "good" views for texturing a polygon [8]. However we define what "good" means, affects the likelihood energy.

The smoothness term E_{smooth} defines the clique potential. Since we want to describe the interaction of two neighboring polygons only the cliques of the second order are considered. By defining the smoothness term we describe how the assignment of different labels to two adjacent polygons affects the seam (i.e the edge between two different images stitched together).

Different implementations of this approach specify E_{data} and E_{smooth} differently. For example, Lempitsky and Ivanov [16] determine the data term based on the angle between the camera direction and polygon normal. This, however, can lead to cases where very distant views can be picked to texture a polygon (proximity of the camea to the polygon is not considered). Allene et al. [21] take the size of projected patch into account, this approach eliminates the problem of distant views but suffers from out-of-focus close-up being picked for texturing. Gal et al. [9] use the gradient magnitude of the source image integrated over the polygon's projection, this eliminates both problems of blurry and small samples [8]. The term is higher if more edges can be identified, hence the assumption is that more details are visible.

The smoothness term can be computed as the itegrated difference in color and intensity of the pixels on two sides of the seam, as it is suggested by Lempitsky and Ivanov [16], but this can become a *"computational bottleneck and cannot be precomputed due to the prohibitively large number of combinations"*, as noticed by [8]. Because of that Waechter, et.al. [8] suggest to define smoothness term using the modified Potts model [22]:

$$E_{\text{smooth}} = [l_i \neq l_q] \tag{2.17}$$

where $[\cdot]$ is an Iverson bracket:

$$[P] = \begin{cases} 1 & \text{if } P \text{ is true} \\ 0 & \text{otherwise} \end{cases} \tag{2.18}$$

This means that the energy is maximal if we have to use different source images for adjacent polygons (we have to do most amount of work). On the other hand, if we use the same image to texture two adjacent polygons, energy in minimal (we don't have to do any work). This simple rule results in labeling, that assign the same image for adjacent faces, getting a priority. This version of E_{smooth} requires no computation and produces very comparable results.

In addition to that, Gal et al. [9] also "expand the space of face labels" and allow for an additional transformation for up to N pixels to be applied to a texture patch projection in the image space. Their labeling is then consists of tuples $f_i = (v_i, t_i)$, where v_i is a source images and t_i is 2D translation vector. This works fairly well in minimizing the seams and misalignment, but additional degree of freedom comes at a very steep computational cost, since

for every pixel of freedom we can have $(2 * n + 1)^2$ possible 2D vectors (in the x and y direction, both positive or negative). For that reason, their approach was not implemented.

To summarize what was chosen for the final tool:

- E_{data}: Gradient magnitude of the source image integrated over the polygon's projection.
- E_{smooth}: Term that favors the same image being picked for adjacent polygons.

Chapter 3

Analysis and Design

In the previous chapter general methods of texture extraction were described and one method (MRF based method) was picked. In this chapter the overall texture extraction pipeline will be outlined. This pipeline directly translates to how the implemented tool operates. After that each step of the extraction process will be discussed in more details, and different general computer graphics techniques used in the tool will be mentioned. And finally couple new ways of solving some minor challenges, involved in the texture extraction process, will be described.

3.1 Pipeline overview

Texture extraction process can be broken downs into the following steps:

- **Data cost calculation**. During this step E_{data} term is calculated.
- **Label assignment via MRF Solution**. Using E_{data} and E_{smooth} terms we calculate the optimal view-to-polygon labeling.
- **Texture data extraction**. Using the labeling from the previous step, texture patches are generated for each polygon.
- **Post-processing**. Generated texture patches are further modified to minimize color discrepancies and mitigate precision errors.

Pipeline is shown in the diagram 3.1.

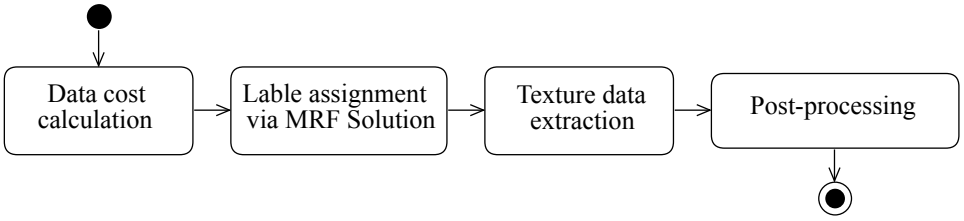

Figure 3.1: Extraction process pipeline.

3.2 Data cost calculation

To calculate the E_{data} term we have to essentially recreate the source images using the known camera position and calibration. This leads to a rasterization process but instead of outputting images onto a screen or a picture we instead need to perform sampling from the source images. And since we defined E_{data} by a gradient magnitude sum, we will be sampling from the source image modified by a Sobel operator [23]. Additionally, as part of data cost calculation step, tool also performs a color consistency check that will be discussed further in this section.

3.2.1 Camera representation

To be able to recreate a source image, we require a very accurate information about camera position and orientation. There are couple of things we have to know about camera to be able to replicate what it sees:

1. Position of a camera in world space coordinates $\mathbf{c} = (c_x, c_y, c_z)^T$.
2. Rotation angles of a camera in world coordinates $(\alpha_x, \alpha_y, \alpha_z)$.
3. Focal length of a camera f_c.
4. Width and height of a bitmap produced by a camera w_c, h_c.

Using this information, we can compute a view space matrix $\mathbf{V}$ and a camera projection matrix $\mathbf{P}$ (the following matricies are based on [24, 25]).

The view matrix $\mathbf{V}$ is derived traditionally as:

$$\mathbf{V} = \left[\begin{array}{c|c} \mathbf{R}_c & \mathbf{c} \\ \hline 0 & 1 \end{array}\right]^{-1} \tag{3.1}$$

$$\mathbf{R}_c = \begin{pmatrix} \cos\alpha_z & -\sin\alpha_z & 0 \\ \sin\alpha_z & \cos\alpha_z & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} \cos\alpha_y & 0 & \sin\alpha_y \\ 0 & 1 & 0 \\ -\sin\alpha_y & 0 & \cos\alpha_y \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\alpha_x & -\sin\alpha_x \\ 0 & \sin\alpha_x & \cos\alpha_x \end{pmatrix} \tag{3.2}$$

To derive the projection matrix we first need to calculate the field of view (FOV) from w_c, h_c and focal length f_c. For our purposes (that are explained in chapter 4) the focal length is represented in pixels so then the FOV ϕ of a camera can be easily computed as:

$$\phi = \arctan\left(\frac{\max(w_c, h_c)}{f_c}\right) \tag{3.3}$$

Additionally we set z_n and z_f to represent near and far planes of the symmetric view frustum, respectively, and the aspect ratio $a_c = w_c/h_c$, so then we can derive the camera projection matrix $\mathbf{P}$ as follows:

$$\mathbf{P} = \begin{pmatrix} \left(a_c \cdot \tan\left(\frac{\phi}{2}\right)\right)^{-1} & 0 & 0 & 0 \\ 0 & \left(\tan\left(\frac{\phi}{2}\right)\right)^{-1} & 0 & 0 \\ 0 & 0 & -\left(\frac{z_f + z_n}{z_f - z_n}\right) & -\left(\frac{2(z_f z_n)}{z_f - z_n}\right) \\ 0 & 0 & -1 & 0 \end{pmatrix} \tag{3.4}$$

Using those we can transform every vertex in a mesh to correspond to its estimated position in a source image.

3.2.2 Image distortion

Another important aspect of recreating a source image is to be aware of lens radial distortions often present in the photographs. Image 3.2 shows two main types of image distortions. Those distortions cause the shift of the actual positions and shape of the objects visible from the camera in the final image. This means that our sampling can be very inaccurate especially for the objects that are closer to the edge of the image.

Luckily the reconstruction tools that our tool supports all perform image "undistortion" as part of their workflow. This allows us not to account for the distortion, and use fixed image provided by the reconstruction tools.

(a) Barrel distortion (b) Pincushion distortion

Figure 3.2: Types of radial distortions. (Source [26])

3.2.3 Triangle visibility

First to simplify things we assume that our model is triangulated; that is, every polygon is a triangle. To determine the polygon visibility first back face and view frustum culling are performed.

To perform back face culling the triangle winding order is utilized. First a cross product of two vectors with origin in the same vertex needs to be calculated. Let's say we have a triangle with vertices A, B, C, and two vectors $\vec{v} = B - A$ and $\vec{q} = C - A$, then the cross product $c_{\triangle} = \vec{v} \times \vec{q}$ can be used to determine if triangle is facing the camera. If triangle's winding order is defined counterclockwise, then triangle is facing the camer only if a cross product is positive $c_{\triangle} > 0$. On the other hand, if the winding order of a triangle is defined clockwise, it is visible only if a cross product is negative $c_{\triangle} < 0$. [27]

For the frustum culling we have to check the vertices of a triangle against every frustum plane. Triangle can considered outside of the frustum (i.e invisible to the camera) only if all of its vertices lay on the wrong side of the same frustum plane. This check is performed in a clip space [28] so then to check a vertex $v = (v_x, v_y, v_z, v_w)$ against all 6 frustum planes we simply check:

- $v_x \geq -v_w$: left plane.
- $v_x \leq v_w$: right plane.
- $v_y \geq -v_w$: top plane.
- $v_y \leq v_w$: bottom plane.
- $v_z \geq -v_w$: near plane.
- $v_z \leq v_w$: far plane.

This way, if we can find a plane for which all vertices will fail the above check, we can consider the triangle to be outside of the frustum. [29]

For this implementation, cases where polygon is only partially in the frustum are not considered for the labeling, since polygon texture is sampled from only one view.

Finally for the pixel-per-pixel occlusion check, the z-buffer is used. Each pixel in z-buffer stores the z value that corresponds to the distance to the camera of the closest triangle seen so far. During the initialization the z-buffer if filled to hold the farthest distance that can be represented. When a triangle is rasterized for each pixel the z value (a.k.a depth value) gets interpolated. And so we only sample pixel from the source image (analogously to writing a value into raster) if our z-value is closer to the camera that what was already in the z-buffer [30]. (How the tool deals with the samples being taken from the area that would later be determined to belong to a different triangle is described in chapter 5)

3.2.4 Triangle lookup

To further speed-up the data cost calculation process we can additionally limit the set of triangles that will be considered for the rasterization. This is achieved by utilizing bounding volume hierarchy (BVH) constructed on our mesh. For the bounding volume representation axis aligned minimum bounding boxes were used (AABB).

For our implementation AABB's are aligned along the word coordinate axes, and each AABB is the smallest possible AABB for a set of primitives (hence the name "minimal" bounding box). Each AABB can be defined by a set of eight vertices or six planes. To construct AABB we first define a root node that encompasses the whole mesh. We then pick the longest axis x, y or z of the current node's bounding box and determine the point δ as the median point on that axis. After that we divide the current node into two children nodes by a partitioning plane orthogonal to the longest axis. This plane goes through the δ and is defined by it. After that the process is repeated recursively for the child nodes, until no further division is possible or the special condition is met. In our implementation this special condition is the size of the node. The smallest node is limited by the number of triangles it has, when that number is reached no further division is performed on that node. [31]

As the result of this process we can easily reject whole sets of triangles if their bounding box is outside the viewing frustum. If the bounding box is completely or partially inside the frustum we can perform the same frustum-visibility check on it's children. And if the node doesn't have any children we then take all of its triangles and considered them for rasterization as usual (performing per-triangle frustum checks and back face culling).

To determine if the AABB is inside the viewing frustum the process identical to the one described in the section 3.2.3. AABB can be considered outside of the frustum if all of its eight vertices lay on the wrong side of the same frustum plane.

Since we only deal with static meshes we can construct the BVH once and use it through the program.

3.2.5 Gradient magnitude

By calculating the data term E_{data} we define what a "good" texture means. We assume that good texture has a lot of details. In computer vision this problem can be interpreted as one of the edge detection. Numerous methods of finding edges can be identified and based on the comparison performed by Maini et.al. [32], the Sobel approach was picked, because it appears to give better result for finding high frequency intensity changes.

For the E_{data} sampling, each source image has to be modified with a Sobel filter [23]. Using two different convolution kernels we can compute G_x,

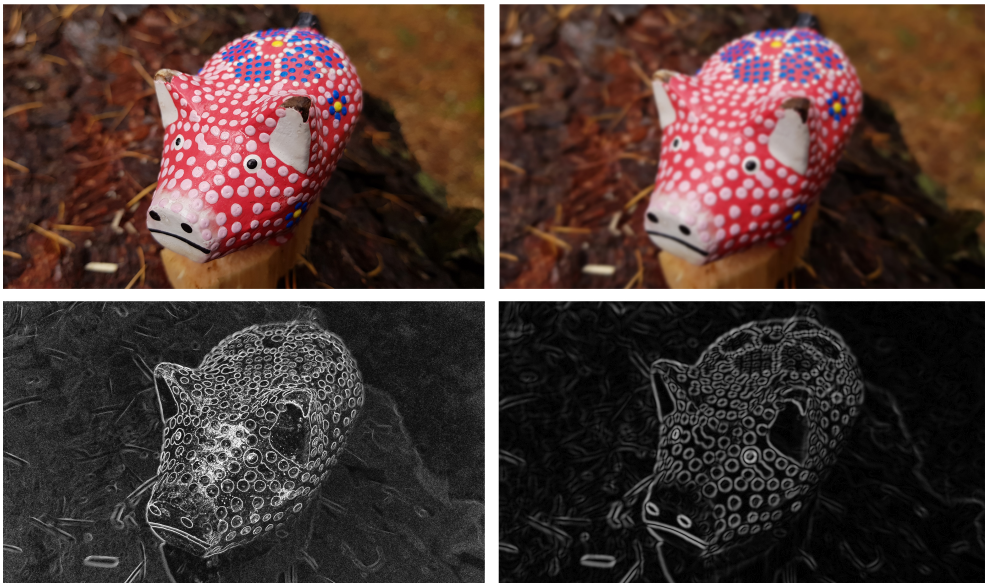

Figure 3.3: Sobel operator applied to the source image. *Top-left*: Original source image. *Top-right*: Source image blurred with Gaussian Blur (radius 10 pixels). *Bottom-left*: Sobel operator applied to original image. *Bottom-right*: Sobel operator applied to the blurred image. (Both Sobel images were equally brightened up for better readability.)

G_y - two oriented gradient components that respond to vertical and horizontal edges respectively. The gradient magnitude G is then computed with $G = \sqrt{G_x^2 + G_y^2}$ [33]

Using edge detection for the data term also allows us to eliminate the problem of blurry textures. Since sharper source images will produce more edges, the respective data term will be more favorable for them. As demonstrated in the figure 3.3, the Sobel operator was applied to the same source image twice, one time to the original and once to the blurred version. As expected the image without the blur produces more edges (which means it will be more likely to be picked for texturing).

3.2.6 Color consistency

After the data cost calculation step is performed, for each polygon we will have a set of candidate views and each view will have a corresponding cost-of-assignment associated with it in relation to that given polygon. Let's denote candidates of the polygon p as a set of tuples $K_p = \{(v_1, e_1), (v_2, e_2), \dots\}$, where v_i signifies some source image candidate and e_i is its calculated data term.

But before we can use this information to get the labeling, the tool per-

forms an additional optional step suggested by [8]. Namely a color consistency check. This step can be useful for large scale reconstructions as it helps to eliminate images with unregistered occludes like pedestrians or source images with a drastically different exposure or color balance. Additionally, during the testing of the tool it was discovered that color consistency check can help with some reconstruction precision errors (more details are provided in the corresponding chapter 6.)

To perform color consistency check, in addition to calculating data costs in the rendering step, we calculate the average color across the polygon of each polygon-view pair. This expands the tuple representing the view candidate to also hold polygon average color information $\overline{m}_i$, making the set of candidates $K_p = \{(v_1, e_1, \overline{m}_1), (v_2, e_2, \overline{m}_2), \dots\}$. For every polygon we then go through all its candidates and calculate the median of all the average colors. Let $\tilde{m}_p$ denote the median average color of the polygon p. Additionally we define a constant $\delta \in (0, 1]$ that signifies the deviation limit from the median. If for the view candidate $k_i = (v_i, e_i, \overline{m}_i)$, the average $\overline{m}_i$ differs from the median average $\tilde{m}_p$ by more than $(1 - \delta) \in (0, 1]$, this candidate gets discarded. In other words discard candidate $k_i = (v_i, e_i, \overline{m}_i)$ if following is true:

$$\left(\frac{\min(\overline{m}_i, \tilde{m}_p)}{\max(\overline{m}_i, \tilde{m}_p)} \right) < \delta \tag{3.5}$$

The higher we set the δ, the stricter the consistency (more candidates get discarded). This approach has its limitation that will be discussed in the later chapter 6.

Additionally Waechter et.al. [8] use a shifting mean instead of a simple median to better refine the consistency check. This method was not implemented in the tool but it can be an interesting point of improvement.

3.3 Label assignment via MRF Solution

As mentioned in the sections 2.3.1 and 2.3.2 we posed the texturing problem as one of the labeling, where the most optimal labeling is a MAP labeling that can be calculated by minimizing the energy term $E(f)$ which consists of two terms, data term E_{data} and smoothness term E_{smooth}.

For the E_{smooth} we picked a modified Potts model, where we essentially set smoothness term to 1 to penalize the usage of different source images for adjacent polygons, and to 0 to encourage the usage of the same image for adjacent polygons.

For our data term E_{data} we calculate the sum of gradient magnitude across the whole visible part of the polygon for every polygon-view pair. Higher gradient magnitude sum means we have detected more edges. First we normalize the sum across all the candidates of one polygon, and since we are looking for energy minimization we have to set data cost of candidate with the lowest

amount of detail to a highest number and vice versa, data cost of candidate with a highest amount of detail to the lowest number. This can be achieved easily for the candidate $k_i = (v_i, e_i, \overline{m}_i)$ of the polygon p, where e_i is already a normalized data term, the following way:

$$E_{\text{data}}(p, v_i) = 1 - e_i \tag{3.6}$$

To get the labeling, by minimizing the energy term, a third-party library mapMAP [34] is used.

3.4 Texture data extraction

After we get the MAP labeling each polygon has one single view assigned to it (or none if the triangle was not visible from any view). To generate the texture map we have to perform forward texture mapping.

To simplify things, we again assume that our model is triangulated, so we only deal with triangles. Each vertex in triangle has a (u, v) coordinate associated with it. (Generation of those coordinates is a task in and of itself and is not within the scope of this thesis). Those UV coordinates are generally normalized $(u, v) \in [0, 1]^2$ and are used for a texture lookup.

Forward texture mapping is a variation of mapping where each texel in a source texture patch is projected towards a destination pixel. (As opposed to, more common, inverse mapping where order is reversed and the lookup is performed for every pixel on the destination render). For our purposes we instead perform sampling from a source image for every texel in a texture patch.

To achieve perspective correct mapping and interpolation, an edge gradient approach described by Chris Hecker [35] was used.

3.5 Post-processing

Texture images generated using the labeling calculated in the previous step can still suffer from color discrepancies between patches, and further adjustment is needed to minimize the seams [8]. Additionally texture generation can suffer from precision errors and that can lead to texture patches being couple pixels smaller along the edges. This can cause the gaps in the final renders. That is why an additional set of modifications is required:

1. Global patch adjustment.
2. Seam leveling.
3. Patch Expansion.

All these steps will be described in more details further.

3.5.1 Vertex samples

Before global and seam leveling can be describe, we have to define something that we called *vertex sample list*.

Since each vertex can belong to more than one polygon and different polygons can have different source images assigned to them, we can say that a vertex can have more than one color associated with it depending on what source image we sample from. (The same is true for the whole edge but for our method we focus on vertices).

Naturally the vertex sample list S_v of the vertex v is a a list of those colors taken from different source images. $S_v = (c_v(j), c_v(m), c_v(n), \dots)$, where $c_v(i)$ denotes the color sample taken from source image i at a vertex v.

For every vertex v we then can calculate mean sample color $\tilde{c}_v$ from samples $(c_v(j), c_v(m), c_v(n), \dots)$. This information will be crucial for solving color adjustment problems.

3.5.2 Global patch adjustment

Under the term "global patch adjustment" we specify the process where the same color adjustment is applied to the whole texture patch associated with the polygon. This step is necessary since only doing seam leveling (which is described next) is often not enough to deal with major color and exposure differences between source images.

For the global adjustment we came up with a relatively simple method. First we perform global adjustment not on a per-polygon texture patch basis but calculate adjustment value for the whole patch set. Patch set is defined as a set of texture patches adjacent to each other in the mesh and also textured with the same source image.

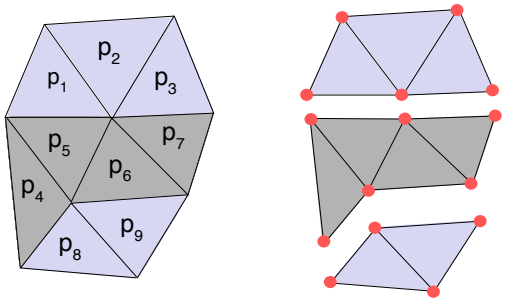

Figure 3.4: A simple mesh. Blue and gray colors signify two different source images being picked for the polygons. *Left*: Mesh structure. *Right*: 3 separate patch sets. Red dots signify *edge sample points* for a given patch.

Let's say that each polygon p_i has a texture patch t_i associated with it. Image 3.4 shows a simple 3D mesh. Different colors signify different source images being assigned to a texture patch. In that example we have two source images; one is represented by a blue color and other by a gray color. We can identify 3 patch sets $P_1 = (t_1, t_2, t_3), P_2 = (t_4, t_5, t_6, p_7), P_3 = (t_8, t_9)$. Even though patches P_1 and P_3 are both textured with the same image they are not adjacent to each other in the mesh (they don't share a common edge).

For every patch set we can identify the *edge sample points*. These are the vertices that lay on the outer edge of the patch set. On the image 3.4 those points are marked with red dots. Patch P_1 has 5 edge sample points, P_2 has 6 and P_3 has 4.

At every edge sample point we can get a mean color using vertex sample list. To get the color, that will be used for the adjustment of a patch set, we then calculate a list of deviations-from-the-mean. Let's say we have a patch set P with edge sample points (verticies) $V = (v_1, v_2, \ldots, v_n)$, list of corresponding source samples taken from the source image i (common for every patch in a patch set) $C = (c_{v_1}(i), c_{v_2}(i), \ldots, c_{v_n}(i))$ and finally list of mean samples from the vertex sample list is $M = (\tilde{c}_{v_1}, \tilde{c}_{v_1}, \ldots, \tilde{c}_{v_n})$. We then calculate a list of derivations-from-the-mean D as:

$$D = (\delta_{v_1}, \delta_{v_2}, \ldots, \delta_{v_n}) = ((c_{v_1}(i) - \tilde{c}_{v_1}), (c_{v_2}(i) - \tilde{c}_{v_2}), \ldots, (c_{v_n}(i) - \tilde{c}_{v_n})) \quad (3.7)$$

And finally to get the color to be applied to the whole patch set we calculate the mean of D.

This process is repeated for every patch set and in the end we get a globally adjusted texture image. The result of this modification will be shown in the chapter 6.

3.5.3 Seam leveling

After we perform global adjustment we still can be left with color seams, because global adjustment only attempts to move adjacent patches closer to the mean and does not achieve total color smoothness.

To perform seam leveling we can utilize vertex sample list yet again. Seam leveling is performed on a per polygon patch basis. First we assume that our polygon is triangle or can be easily triangulated. Then for all three vertices of a triangle we calculate the derivations-from-the-mean in a similar fashion to what was described in the previous section 3.5.2. For the triangle $\triangle ABG$ that was assigned a view i we have a list of derivations-from-the-mean $D_{ABC} = (\delta_A, \delta_B, \delta_C) = ((c_A(i) - \tilde{c}_A), (c_B(i) - \tilde{c}_B), (c_G(i) - \tilde{c}_G))$. To perform seam leveling we interpolate the derivation-from-the-mean δ across the whole texture patch using $\delta_A, \delta_B, \delta_C$ and add it to the original color. This helps a lot with smoothing of the seams.

Unsurprisingly if the vertex is only a part of the polygons that were assigned the same source no adjustment is performed. Additionally it is important that the severity of this adjustment is directly proportional to the size of the polygon since the change is not propagated beyond the polygon. This means that larger polygons will be affected more than the smaller ones.

It is also important to note that after the global adjustment (and before the seam leveling) we have to update the vertex sample list to take its effect on color into account. Result of this adjustment is shown in the chapter dedicated to testing 6.

Alternatively [9] suggest to use Poisson blending to minimize the seams. This method was not implemented but can definitively be considered as a possible future improvement.

3.5.4 Patch Expansion

Patch expansion is the simplest modification of three and only involves stretching of the edge pixels of the polygon outwards to expand its size.

This step was found to be necessary during testing, because precision errors resulted in the gaps in the final renders.

3.6 Independent processing steps

To satisfy the requirement **F7: Independent processing steps** outlined in section 1.1, major computational sections have to be identified, so we can save the progress between those steps.

Naturally those sections are almost identical to the steps of the texture extraction pipeline. Three major sections are:

1. Data cost calculation
2. Color consistency check and label assignment
3. Texture extraction and post-processing

After the data cost calculation step we can save all gradient magnitude sums and color mean information of all the view-polygon pairs. This information is saved before the color consistency check, because this allows the user to modify deviation limit δ to get a more strict or more lean color check without having to rerun the data cost computation.

After the color consistency and label assignment we save the optimal labeling that defines the source image assignment. This step is separate from post-processing because it allows us to use the same labeling and modify the post-processing configuration. This allows user to, for example, opt-out of performing a certain modification (like seam leveling).

This means that the intermediate process is saved twice. Once for the data costs and later for labeling.

Chapter 4

Technologies

In this chapter libraries and tools used in the development will be described.

4.1 Libraries

In accordance with the requirement **S1: Portability** (1.1) all of the libraries are operating system independent.

4.1.1 mapMap

mapMAP is a massively-parallel, generic, MRF MAP solver that allows for a rapid solution of a large class of MRF problems [34]. mapMAP is implemented as a templated, header only library, and was created as part of a 2016 paper by Daniel Thuerck et al. [36]

In our tool we use this library to get the view assignment labeling by passing it the E_{data} and E_{smooth} terms.

4.1.2 OpenCV

For manipulating with images, like applying the Sobel operator, we used OpenCV[5] library. OpenCV is an open source library capable of solving hundreds of different computer vision and machine learning problems like faces recognition, images stitching, eye movements tracking, etc. [37]

4.1.3 GLM

OpenGL[6] Mathematics (GLM) is a header only mathematics library for graphics software [38]. This library was primarily used for matrix and vector calculations.

[5] Open Source Computer Vision Library
[6] Open Graphics Library

4.1.4 inih

For the configuration of the tool an INI file format [39] was used. To parse the configuration file, we used a header only library – inih [40].

Listing 4.1 shows a snippet of an INI file example used in our tool.

```
[basics]
objFilePath         = path/to/model.obj
cameraListFilePath = path/to/camera/list.txt
cameraInfoPath      = path/to/camera/info/bundle.rd.out
photoFolderPath     = path/to/photo/folder
...
[optional]
doGloabalAdjustment = true
doSeamLeveling       = true
doTextureExtension  = false
...
```

Listing 4.1: INI file example

Each entry in the INI file belongs to a specific category (in the example above we have two categories: ”basics” and ”optional”). inih parser supports integers, real numbers, booleans and strings.

4.2 File formats

4.2.1 Bundle

Bundle format is a format used by all three supported reconstruction tools VisualSFM [5], Bundler SFM [4] and laser 3D scan developed in the CTU FEE Department of Computer Graphics and Interaction [6]. It is generated during the reconstruction step and contains the camera information for the source images.

Listing 4.2.1 shows a bundle format taken from [4].

Since are not interested in the points , they are not going to be discussed. Variable `<num_cameras>`, naturally, signifies a number of cameras (source images) we have. Each camera entry consists of five entries. For us the important ones are:

- `<f>` : the focal length in pixels.

  (This corresponds to f_c from the section 3.2.1.)

- `<R>` : a 3x3 matrix representing the camera rotation.

  (This corresponds to the matrix $\mathbf{R}_c$ from the section 3.2.1.)

- `<t>` : a 3-vector describing the camera translation.

  (This corresponds to the **c** from the section 3.2.1.)

Additionally it was noticed by [41] that Bundler version uses the left handed coordinate system for the camera space and to convert to a more traditional right handed system (used in OpenGL) we have to flip z and y axis for both rotation and translation.

```
# Bundle file v0.3
<num_cameras> <num_points>
<camera1>
<camera2>
. . .
<cameraN>
<point1>
<point2>
. . .
<pointM>
```

Each camera entry `<camera>` contains the estimated camera intrinsic and extrinsic parameters, and has the form:

```
<f> <k1> <k2>
<R>
<t>
```

Listing 4.2: Bundle file format.

Bundle file does not contain a reference to a source image files, instead it is associated with another file that is generated by Bundler or VisualSFM. This file is a simple list of all source images and their order in that list corresponds to the camera info entries in the bundle file. This is why our tool requires both of those files as an input.

4.2.2 Wavefront OBJ

Wavefront OBJ [7] is one of the standard file formats commonly used to define 3D geometry and includes the position of each vertex, information about how polygons are formed in the geometry, UV coordinates and normals. Format is also often expanded to support multiple objects per file.

Each vertex has an identification number associated with it and is given by its order in the file as it appears. Each texture coordinate and normal also has a corresponding ID number associated with them. This allows for them to be defined once and reused multiple times in many polygons by referencing to them by their identification numbers.

4.3 Tools

4.3.1 VisualSFM

For generating most of test data we used VisualSFM [5]. This is a photogrammetry tool that takes a number of images and outputs both camera bundle information file and a dense point cloud reconstruction in a PLY[7] format.

4.3.2 Meshlab

Meshlab [42] is a very feature-rich open source system for processing and editing 3D triangular meshes. Meshlab was used to convert dense point cloud PLY models into a reconstructed OBJ models, but also for the simplification of the model and initial cleanup.

4.3.3 Blender

To generate UV maps (UV coordinates) and perform mesh cleanup we used another free and open source 3D software – Blender [43].

[7]Polygon File Format, also known as Stanford Triangle Format.

Chapter 5

Implementation

In this chapter the most interesting and important parts of the implementation will be described.

5.1 Mesh representation

Our mesh representation is designed to mimic an OBJ file format. This is done intentionally to simplify the reading from the file and argument passing, since every element on the mesh can be referenced by its ID.

```
1  class Mesh {
2      std::map<uint, Vertex>   verticies;
3      std::map<uint, TexCoord> texCoords;
4      std::map<uint, Triangle> triangles;
5      std::vector<Object> objects;
6          . . .
7  };
8
9  struct BoundingBox{
10     glm::vec3 minVec;
11     glm::vec3 maxVec;
12         . . .
13 };
14
15 struct Object{
16     BoundingBox boundingBox;
17     std::string name;
18     std::vector<uint> triangles;
19     PartitionNode partitionRoot;
20 };
21
```

```
22 struct PartitionNode{
23     PartitionNode * leftNode  = nullptr;
24     PartitionNode * rightNode = nullptr;
25     PartitionNode * parent    = nullptr;
26     std::vector<uint> triangles;
27     BoundingBox boundingBox;
28     float separator;
29     Direction direction = NONE;
30 };
```

Listing 5.1: Mesh structure.

Using the `PartitionNode` we build the BVH mentioned in the 3.2.4. Variable `direction` signifies by which axis plane the bounding box is separated (x, y or z). Variable `separator` corresponds to δ from the 3.2.4.

Each bounding box is defined by six axis-aligned planes that are stored as two vertices: `minVec` $= (min_x, min_y, min_z)$ and `maxVec` $= (max_x, max_y, max_z)$.

5.2 Patch Quality representation

`PatchQuality` is a structure that hold all the information necessary to compute the data term E_{data}. It is associated with one triangle and one source image(view).

```
1 struct PatchQuality {
2     uint sampleCount           = 0;
3     uint potentialSampleCount  = 0;
4     float gradientMagnitudeSum = 0;
5     glm::vec4 colorSum;
6 };
```

Listing 5.2: PatchQuality structure.

For the `PatchQuality` we keep track of the following parameters:

- `sampleCount` : a number of pixels that were sampled for this triangle during the rasterization phase. It gets increased every time a pixel is sampled, it gets decreased if that pixel is later determined to belong to another triangle (due to occlusion).
- `potentialSampleCount` : a number of pixels a triangle could have potentially had from a view regardless of occlusion (only gets incremented). If a strict occlusion is picked in the configuration then view-triangle pair gets discarded if a `sampleCount` $\neq$ `potentialSampleCount`.

- `gradientMagnitudeSum` : a total color sum of samples taken from the source modified by Sobel operator. In the same fashion as `sampleCount` gets decreased if sample is later determined to belong to another triangle.
- `colorSum` : a color sum, sampled from the original source image. Same decrease/increase principle is applied to this variable as well. Used for the color consistency check (3.2.6).

(Code snippet 5.4 [lines 10–15] in the next section shows the incrementation and decrementation of `sampleCount`, `gradientMagnitudeSum` and `colorSum`)

5.3 Data costs extraction

Since data cost extraction consists of independent steps, each one related to one source image, we can perform it in parallel. We first divide the source image workload equally between threads and then process a set of images on each thread simultaneously.

A `DataCostExtractionManager` class is responsible for keeping track of the workload and performs the main thread loop in the `doWork` method, and a `DataCostsExtractor` class perform an extraction for one source image itself.

```
1  class DataCostExtractionManager{
2      std::map<uint,std::map<uint,PatchQuality>> * dataCosts;
3      std::vector<View *> viewsToDo;
4      void doWork();
5      . . .
6  };
7
8  void DataCostExtractionManager::doWork(){
9      for(auto v : viewsToDo){
10         View & view = *v;
11         DataCostsExtractor extractor(mesh, view);
12         (*dataCosts)[view.id] = extractor.calculateCosts();
13     }
14 }
```

Listing 5.3: DataCostExtractionManager.

`DataCostsExtractor` method `calculateCosts` returns a set of data costs for every triangle visible from one particular source image. Those sets are combined into one `dataCosts` set local to a `DataCostExtractionManager`. This means, that since we have as many `DataCostExtractionManagers` as threads, we have to later merge them.

To perform data cost extraction we used scanline rasterization and edge gradients described here [35]. To help with the occlusion `DataCostsExtractor`

has two buffers: a z-buffer (`depthBuffer`) and an ID buffer that keeps track of what triangle is assigned to what pixel (`idBuffer`). The second buffer `idBuffer` is used to help with sampling from areas that can later be determined to belong to another triangle as mentioned in the section 5.2.

Here is a snippet of code that deals with one scanline:

```
1  void DataCostsExtractor::drawScanLine(Edge left, Edge right,
   ↪  int y, Gradient & gradient, uint id){
2     . . .
3      for(int x = xMin; x<xMax ; x++){
4          int index = x + y*width;
5          patchInfos[id].potentialSampleCount++;
6          if(depth < depthBuffer[index]){
7              glm::vec4 color = sobelImage.at(x, y);
8              glm::vec4 originColor = sourceImage->at(x,y);
9
10             if(idBuffer[index]!=0){
11                 uint previousID = idBuffer[index];
12                 patchInfos[previousID].gradientMagnitudeSum -=
                   ↪  color[0];
13                 patchInfos[previousID].sampleCount--;
14                 patchInfos[previousID].colorSum -=
                   ↪  originColor;
15             }
16
17             if(id != 0){
18                 patchInfos[id].sampleCount++;
19                 patchInfos[id].gradientMagnitudeSum +=
                   ↪  color[0];
20                 patchInfos[id].colorSum += originColor;
21             }
22
23             idBuffer[index] = id;
24             depthBuffer[index] = depth;
25         }
26         depth += depthXStep;
27     }
28 }
```

Listing 5.4: Scanline calculation.

In the omitted part of the snippet 5.4 (right at the method start) we calculate the `depth` and the `depthXStep` for the whole scanline that goes from `xMin` to `xMax` on the `y` level. Using those we can compute the depth for every

pixel. On the line 6 we first check the depth buffer, and, if the current depth is occluded by something that is already there, we skip. Variable `id` is an ID of the triangle we are rasterizing.

On the lines 10–15 we first check if we already sampled from this pixel by looking at the ID buffer. If we did, it means that the previous sampling was incorrect and we have to accommodate for that by subtracting something that was added before. On the lines 17–21 we add to the patch quality sums. And finally on the lines 23 and 24 we write to depth and ID buffers.

5.4 Computational checkpoints

As detailed in the section 3.6 we have to save two intermediate computations:

1. data costs,
2. labeling.

To save those, simple text based file formats were used.

5.4.1 Data costs file format

```
<num_triangles> <num_triangles_with_data>
<triangle1>
<triangle2>
. . .
<triangleN>
```

Each triangle entry `<triangle>` consists of data cost for that triangle calculated from different cameras:

```
<triangle_id> <num_of_entrise>
<camera1_ID> <magnitudeSum> <sampleCount> <colorSum[3]>
<camera2_ID> <magnitudeSum> <sampleCount> <colorSum[3]>
. . .
<cameraM_ID> <magnitudeSum> <sampleCount> <colorSum[3]>
```

Listing 5.5: Data costs file format.

Since some triangles can still be left unseen by every camera, two separate variables `<num_triangles>` and `<num_triangles_with_data>` are introduced. They are used for validation when loading from file. Entries with `sampleCount` $= 0$ are obviously not saved. Variable `<colorSum[3]>` is a color sum of the triangle represented by 3 entries for all 3 channels (red, green and blue). Listing 5.8 shows an example of the data cost file.

5.4.2 Labeling format

Labeling format is even simpler:

```
<triangle1_ID> <view_ID>
<triangle2_ID> <view_ID>
. . .
<triangleN_ID> <view_ID>
```

Listing 5.6: Labeling format.

For every triangle we save what view (source image) was assigned to it. For the triangles that were left unseen we assign a view ID of 0. That ID is reserved since the actual view IDs start with 1. Listing 5.7 shows an example of the labeling file.

```
1 33
2 33
3 33
4 52
5 33
. . .
7516 35
7517 5
```

Listing 5.7: Labeling format example.

```
7517 7396
1 4
2 6.0902 31 4.01569 17.6824 13.1569
3 1.37647 64 10.3647 7.95294 7.13334
4 4.85491 194 66.6783 42.0236 34.0941
5 3.00392 187 79.9217 45.153 32.7372
2 15
2 6.17255 33 5.76079 21.6392 16.2902
3 1.2 59 9.61177 7.66667 6.71373
. . .
7517 1
5 25.6471 499 31.2784 201.944 107.427
```

Listing 5.8: Data cost file format example.

Chapter 6

Testing

In this chapter the testing of the tool is described. We will first describe how the testing was performed. We then will mention one or two bugs that were discovered during testing and how they were solved. In the main part of this chapter we will discuss how different settings of the configuration affect the performance of the tool and the quality of the final texture. The limitations of the approach used will also be described.

6.1 Setting up the testing

For the testing purposes we used datasets provided by the supervisor as well as the couple we produced ourselves.

To generate test data, sets of photographs were taken using the Samsung Galaxy S8+ 12-megapixel camera [44]. Using the VisualSFM tool, the dense point clouds were generated. We then used Meshlab to generate reconstructed models (we used the "Screened Poisson Surface Reconstruction"). We also used Meshlab to simplify the models for our testing since reconstructed models can have millions of triangles, and for our datasets that was more than necessary. Finally we used Blender to automatically generate UV maps for our objects.

All test we performed on Apple MacBook Pro with a 2,3 GHz dual-core Intel Core i5 processor and 16 GB of RAM.

6.2 Bugs identified during testing

6.2.1 BVH construction failure

When performing one of the tests the BVH construction failed by entering the infinite loop while attempting to divide a certain AABB. In our implementation when triangles are divided between child AABBs their membership is

determined by comparing one single maximal vertex of that triangles bounding box and a separator. Separator is picked using the longest axis (in the code snippet 6.1 the longest axis for the parent node is x).

```
1  void Mesh::constructNode(PartitionNode * node){
2      if(node->triangles.size() <= arguments.BVHMinNode)
3          return;
4  . . .
5          node->separator = ( node->boundingBox.maxVec.x +
           ↪  node->boundingBox.minVec.x)/2;
6          for(auto t : node->triangles){
7              if(triangles[t].boundingBox.maxVec.x <
               ↪  node->separator){
8                  node->leftNode->addTriangle(triangles[t]);
9              }else{
10                 node->rightNode->addTriangle(triangles[t]);
11             }
12         }
13 . . .
14     constructNode(node->leftNode);
15     constructNode(node->rightNode);
16 }
```

Listing 6.1: AABB separation.

This can lead to edge cases where all triangles end up in one group and recursive separation is not stopped, since condition on line 2 in code snippet 6.1 is never met. The solution, naturally, was to stop recursive division if all triangles end up in one group.

The image 6.1 shows such example. Since the initial node is longer in the x direction, a separator is picked accordingly. Because we do not cut the triangle that overlap with separator, all triangles will lay on the same side of the separator after the division.

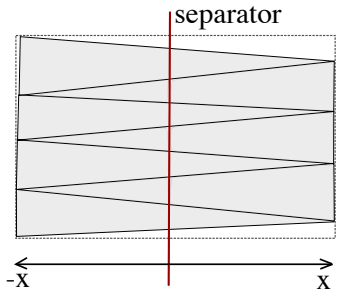

Figure 6.1: AABB generation failure case.

Another solution would have been to try and separate a box by another axis (x, y or z) and stop separation if all of them fail.

6.2.2 Unusual mesh back face culling

When working with another data set, the shape of the model itself caused the problem, where triangles that that were not facing the camera (based on the winding order) still had to be considered for the occlusion of other triangles.

This bug/issue was solved by not skipping the triangle that is facing away from the camera during the rasterization / data cost extraction step, but passing it with an ID equal to 0. Zero ID is reserved for that purpose. No data cost calculation is performed for triangles with ID = 0, but their depth is still recorded in the depth buffer. In the code snippet 5.4 condition on the line 17 takes care of that.

Image 6.2 shows an example of this problem. Image shows a reconstruction of the archaeological site. Since inside walls are facing away from the camera, with traditional culling they should not be considered for occlusion. This can however lead to area marked in red on a source image being used to texture the triangles on the inside of the reconstruction site.

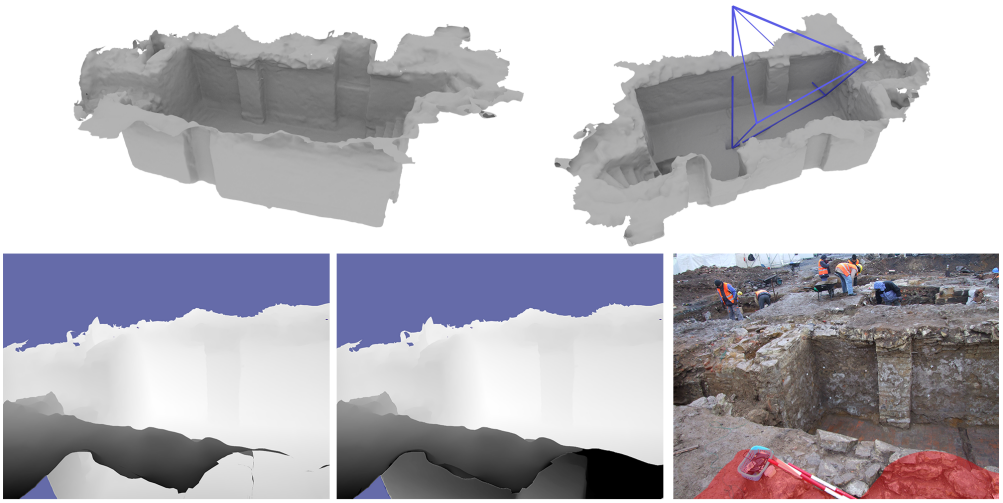

Figure 6.2: Culling error. *Top row left:* reconstructed model. *Top row right:* approximation of the camera position (blue pyramid signifies the position and direction of the camera). *Bottom left:* depth rendering of the source image with traditional back face culling. *Bottom middle:* depth rendering of the source image with modified back face culling. *Bottom right:* Source image. Area in red marks the problematic area.

6.3 Texture Quality

In this section we will discuss how different settings to the tool configuration affect the texture quality and what are the limitations of some of the approaches that were picked .

6.3.1 Color consistency

As part of texture extraction pipeline our tool performs color consistency check. As mentioned in the section 3.2.6 we can control the strictness of the the check by changing the deviation limit δ. In our configuration INI file this option is controlled by the `colorConsistencyThreshold` setting.

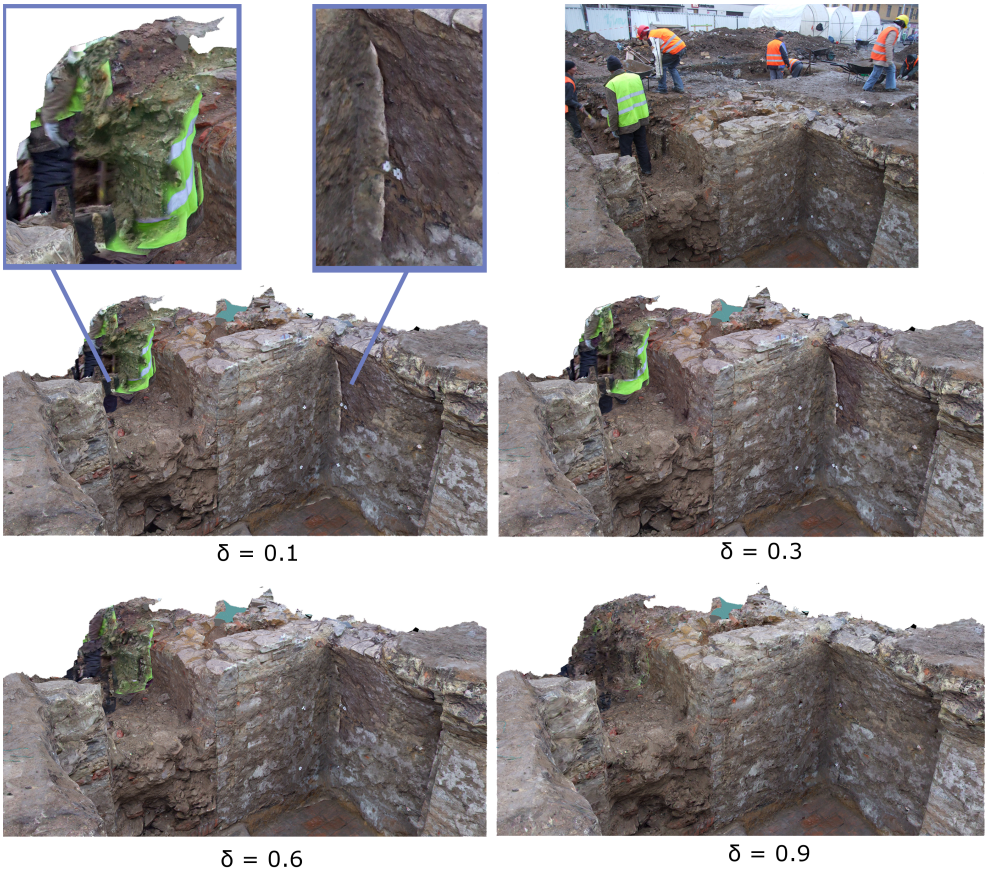

Figure 6.3: Color consistency example. *Top row left:* Close up of the color inconsistency caused by the moving workers present during the reconstruction. *Top row middle:* Inconsistency caused by the reconstruction precision error. *Top row right:* One of the source photographs. *Middle and bottom rows:* Renders of the reconstructed model using the textures generated with varying color consistency thresholds.

δ	Removed view-polygon pairs
0.1	7
0.3	772
0.6	23266
0.9	515288

Table 6.1: Color consistency pair elimination.

Image 6.3 shows an example where dataset had some moving workers present during the collection of source photographs. Without the color consistency check there are some places in the final texture where green west of the worker is clearly visible. After enabling the color consistency check those sections are mostly removed. Unfortunately though in some places the green is still visible. It is mainly due to the fact that some triangles didn't have too many view candidates where the worker in green vest was not present, so the tool was not able to identify those as outliers.

Table 6.1 shows how many view-polygon pairs were removed for the model of around 130 000 triangles and 55 source images, based on the color consistency threshold level (δ) (same dataset as in the picture 6.3 was tested).

Added benefit of doing color consistency is that it can helps with mitigation of the impact of small reconstruction errors. Image 6.3 shows that effect as well.

Unfortunately, one limitation of this approach is that a stricter consistency check can lead to a situation where mostly blurry source patches are left as candidates, since blurry and distant samples are often closer to the mean than the sharper ones. Image 6.5 shows the result of increasing the color consistency strictness. Higher deviation limit produces noticeably blurrier results in certain areas.

Figure 6.4: Color consistency strictness limitation.

6.3.2 Global color adjustment and seam leveling

Image 6.5 shows three pictures:

- **Raw texture**. Rendering of the reconstructed model with a "raw" texture applied to it. "Raw" texture is a texture that was generated by only extracting the texture data from source images based on the labeling and not modifying it further
- **Global adjustment**. Rendering of a model with a texture that was modified by applying global color adjustment to the texture patches.
- **Seam leveling**. Same models but with texture that was additionally improved by performing seam leveling.

A significant improvement in quality after performing global and seam leveling is noticeable.

Figure 6.5: Three stages of texture processing. (Each stage has two close-ups shown.) *Left column:* Rendering using raw texture. *Middle column:* Using texture after global adjustment. *Right column:* Using final texture after seam leveling.

Our approach however is not perfect. Since we calculate the adjustment based on the samples taken from single points on source images, misalignment between couple source images can cause the situation where samples are taken from entirely different points on a object. This leads to artifacts and color bleed. This is unfortunately a common problem in many implementations and is mainly caused by reconstruction errors. Image 6.6 shows an example of those effects. One solution that was proposed by [8] involves calculating the samples as a weighted average along the edge. Their approach was not implemented, but can definitely pose an interesting area of improvement.

Figure 6.6: Seam leveling color bleed.

This error is mostly noticeable in reconstructions that involve small graphic details of significantly different colors (like toys or anything that involves text). Experimentally it was determined that the best approach for those objects is to reduce the number of polygons so that most of the object area will be covered by big uninterrupted parts of same source images.

6.3.3 Precision errors and texture resolution

Very complex models can have millions of triangles which makes it very hard to fit all the texture information on one image file. Of course one could generate a texture image with width and height of couple of hundreds of thousands of pixels but this can become extremely unwieldy. Keeping all the texture information in one file presents another challenge. As mention in the section 3.5.4 precision errors during the texture generation can cause the texture patches be a couple of pixels smaller on the edge than required. The lower the resolution of the texture images the more significant are the errors. During testing we discovered that this produces noticeable gaps in the final renders. This is why we had to add patch expansion step.

Image 6.9 shows a render before and after the texture patches were expanded, as well as the example of the texture image before and after the expansion. Render with texture without the expansion shows very noticeable gaps (white pixels).

Another solution that we came up with, was to allow the user to separate the model into several parts and then generate separate textures for each of them. This allows user to have as many high resolution texture images as they wants for their model. This was the main reason for allowing meshes with several separate objects in them.

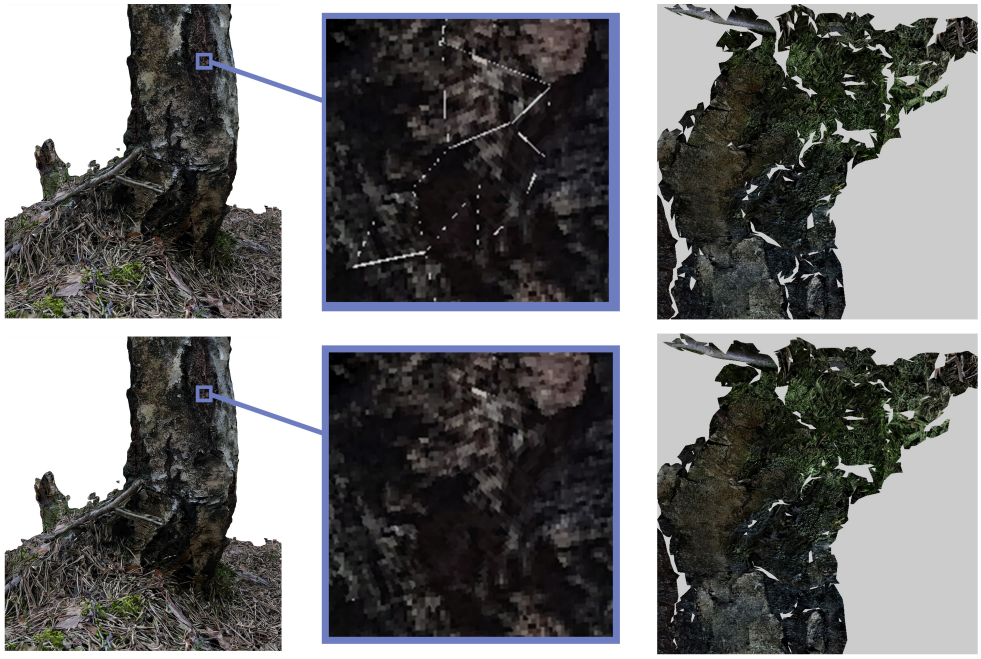

Figure 6.7: Texture expansion. *Top row:* Render, close up of a model and a texture itself before the expansion. *Bottom row:* Render, close up of a model and a texture itself with the expansion performed.

6.4 Performance

To measure the performance we wanted to see how our addition of BVH affected the data cost calculation stage, as well as how different parts of the calculation were affected by such factors as mesh size (i.e polygon number) and number of source images. We also wanted to measure how our multithreading affected the speed of calculations.

Since no strict requirement were posed on the performance, those optimizations and testing were done purely to identify computational bottlenecks and identify points of possible future improvement.

6.4.1 BVH impact

For this test we first performed the data cost calculation step without building a bounding volume hierarchy and measured the time we got. We then set each smallest bounding volume to contain about .1% of all triangles (e.g for a mesh with 200 000 triangles each smallest volume contains 200 triangles). And finally we set smallest volume to contain only one triangle, effectively constructing a complete BVH.

Minimal BVH node is controlled by the `BVHMinNode` variable in the configuration file.

Dataset name: ”Slany archaeological site”.
Mesh size: 129 000 triangles.
Source Images: 54 (3488 x 2616 pixels each)

Minimal node size	Mesh initialization time (includes BVH construction)	Extraction wall time	Wall time to base time (lower means better)
full mesh	0m:09s:648	6m:48s:448	**100.0%**
200	0m:10s:817	4m:44s:724	**69.6%**
1	0m:11s:565	4m:50s:861	**71.1%**

Dataset name: ”Bird toy”.
Mesh size: 125 000 triangles
Source Images: 36 (4032 x 3024 pixels each)

Minimal node size	Mesh initialization time (includes BVH construction)	Extraction wall time	Wall time to base time (lower means better)
full mesh	0m:10s:977	04m:20s:11	**100.0%**
200	0m:12s:88	02m:46s:527	**63.8%**
1	0m:12s:967	04m:04s:58	**93.8%**

Table 6.2: BVH impact on performance.

Test was performed on two meshes. One of an archaeological site and one of a little bird toy. Main difference is that the archaeological dataset had many source images that show only small localized parts of the whole mesh due to the nature of its size in real life. The toy dataset, on the other hand, contained images that all mostly show the object completely.

All tests were performed using 4 working threads.

Table 6.2 shows the result of our testing. It is noticeable that addition of the BVH sped up the data cost calculation step considerably at a relative to no cost for construction.

It is also noticeable that usage of the BVH on the ”Bird toy” dataset did not yield such an increase in performance when setting the `BVHMinNode` to a very small number. As mentioned before the ”Bird toy” dataset consists mainly from source images depicting the toy fully, this means that most of the bounding boxes will always lay inside the view frustum, hence no performance increase from culling.

6.4.2 Multithreading

For this test we wanted to see how significant of an impact on the performance would a multithreading yield on a data cost calculation step. We ran the test 4 times with 1, 2 and 4 threads. Dataset consisted of the mesh with about 129 000 triangles and 54 source images 3488 x 2616 pixels each.

threads	Data cost extraction time		Wall time in seconds	Wall time to base time (lower means better)
	CPU	Wall		
1	10m:47s:680	11m:00s:284	660	**100.0%**
2	10m:56s:470	6m:06s:663	366	**55.5%**
4	17m:00s:960	4m:46s:261	286	**43.3%**

Table 6.3: Multithreading performance text.

Table 6.3 shows the result of our testing. As expected since data costs computation is linear in regards to the number of views, their resolution and a mesh size, the speed up is almost linear at first.

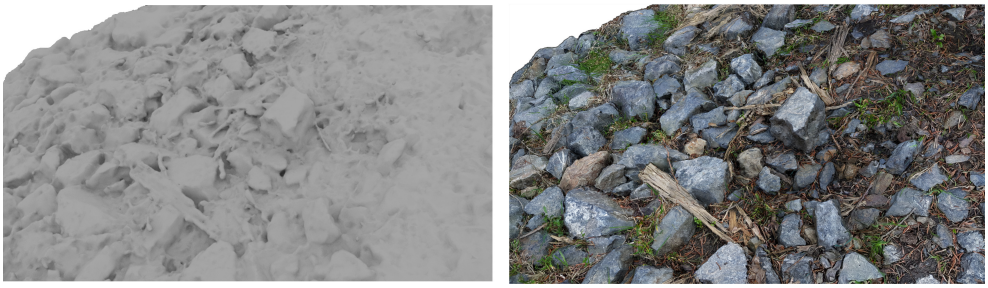

Figure 6.8: Some texturing results. "Ground" dataset

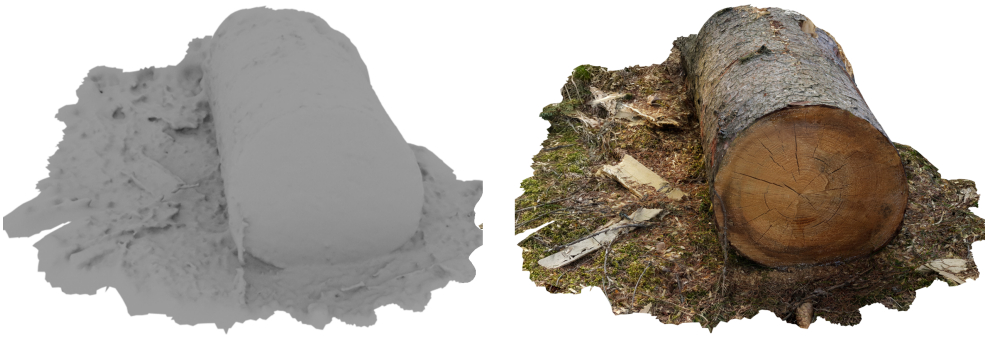

Figure 6.9: Some texturing results. "Stump" dataset.

Name	Mesh size (triangles)	Source images	Final texture
Slany	647 000	54 (3488 x 2616 px)	5 images (4000 x 4000 px)
Ground	112 000	64 (4032 x 3024 px)	1 image (8000 x 8000 px)
Birch	73 000	176 (4032 x 3024 px)	1 image (5000 x 5000 px)

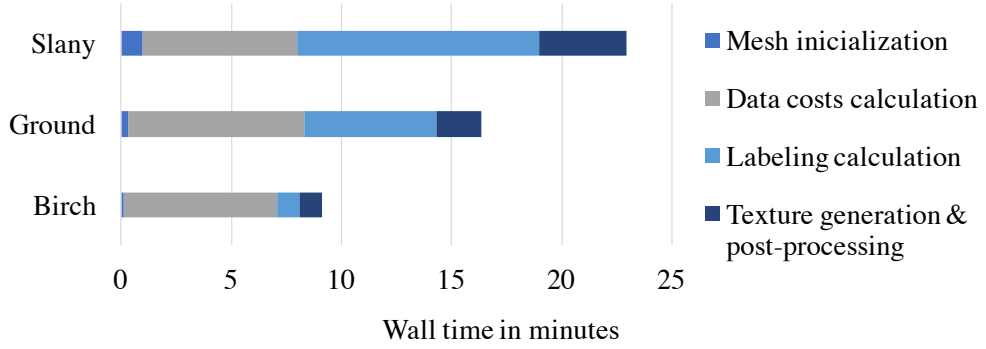

Table 6.4: Tool performance.

6.4.3 Overall performance

To evaluate overall tool performance we tested the tool with 3 datasets. Each test was performed on 4 working threads and an estimated optimal `BVHMinNode`. Table and the graph 6.4 show the result of our testing.

It is noticeable that labeling and especially data costs extraction are the most time consuming steps. Label generation is performed by a third party library that is already supposed to have state of the art performance [36]. Data cost calculation is also already parallelized and we took several steps to increase its efficiency (e.g. BVH frustum culling).

One possible way we can increase the performance is be to perform the computation on the GPU instead of the CPU. This option was out of the scope for this version of the tool, and the modifications to the inner working of our tool, that this switch would entail, were not fully researched.

Chapter 7

Future Improvements

Though the tool is fully functional and satisfies all the requirement posed on it, there are still several areas that could be improved.

7.1 Texture quality

Our approach to color balancing the texture patches and seam leveling produces decent results and does not require heavy computations, but it can still be improved. As mentioned in the previous chapters, we could use Poisson blending for the seam leveling. We also could use weighted color average edge for the vertex color sample list, that we use for seam leveling and global adjustments, this could eliminate color bleed and color artifacts.

7.2 Texture map generation

It would be useful to be able to generate texture map algorithmically within the tool itself. This will allow for a smoother pipeline between the reconstruction tools and our tool. This area presents an interesting problem in and of itself, but was not within our scope for this version.

7.3 Performance

As mentioned before our computation is performed on the CPU. It could be a significant improvement to our performance to compute data costs on the GPU, due to the nature of its process, because data cost computation basically mimics the traditional rasterization.

This option was not within the scope for this version of the tool, and was not researched properly.

7.4 GUI

Though it was established from the beginning of the development that the tool didn't require the GUI, addition of an even very simple GUI would make the tool much more user friendly.

7.5 File format support

Our tool supports one of the most common file formats for 3D models – OBJ file format. Another very common format, especially in several reconstruction tools is PLY [45]. Addition of PLY support will again streamline the pipeline between the reconstruction tool and our tool.

Conclusion

The goal of this thesis was to analyze the process of extracting texture information from photographs used for 3D model reconstruction and implement a tool capable of texture exaction.

First the state of the art texture extraction techniques were described and compared. An approach that was implemented in the tool was determined. The texture extraction process was posed as an image stitching problem and boiled down to a task of assigning each polygon a single photograph to get the texture from. Necessary theory required to solve this problem was presented, namely Markov Random Field modeling. Variations on the picked technique were described after that.

In the next chapter the texture extraction pipeline was more concretely outlined. The process of data cost calculation, necessary for the extraction, was described in detail. A binary space partitioning technique and its application in speeding up the process of rasterization was discussed. A new approach of dealing with color correction of the texture patches was suggested.

Finally the tool was implemented and tested. Multiple datasets were used to test the functionality of the tool and identify some bugs and points of improvement. Performance of the final texture was measured. Limitations of the tool were discussed.

Overall, I'm happy with the result, even though the final tool still can be worked on. Many ideas on how to improve the tool were provided in the dedicated section. It was my first time working with computer graphics and computer vision, so naturally i have learned a lot in the process, for which I am grateful.

Bibliography

[1] 80 level. The Production of 'Kingsglaive: Final Fantasy XV'. [online], october 2016, [cit. 2018-04-22]. Available from: https://80.lv/articles/the-production-of-kinglgrave-final-fantasy-xv/

[2] Electronic Arts Inc. How we used Photogrammetry to Capture Every Last Detail for Star Wars™ Battlefront™. [online], may 2015, [cit. 2018-04-22]. Available from: http://starwars.ea.com/starwars/battlefront/news/how-we-used-photogrammetry

[3] Eeles, P. Capturing architectural requirements. *IBM Rational developer works*, 2005.

[4] Noah, S. Bundler: Structure from Motion (SfM) for Unordered Image Collections. [software], 2010, version 0.4 [cit. 2018-02-05]. Available from: http://www.cs.cornell.edu/~snavely/bundler/

[5] Wu, C. VisualSFM: A visual structure from motion system. [software], 2011, v0.5.26, [cit. 2018-04-19]. Available from: http://ccwu.me/vsfm/

[6] Beránek, P. *Systém pro automatické snímání pomocí 3D laser scanneru a fotoaparátu*. Master's thesis, České vysoké učení technické v Praze, 2017.

[7] Reddy, M. Object Files. [cit. 2018-04-21]. Available from: http://www.martinreddy.net/gfx/3d/OBJ.spec

[8] Waechter, M.; Moehrle, N.; et al. Let there be color! Large-scale texturing of 3D reconstructions. In *European Conference on Computer Vision*, Springer, 2014, pp. 836–850.

[9] Gal, R.; Wexler, Y.; et al. Seamless montage for texturing models. In *Computer Graphics Forum*, volume 29, Wiley Online Library, 2010, pp. 479–486.

[10] Grammatikopoulos, L.; Kalisperakis, I.; et al. Automatic multi-view texture mapping of 3D surface projections. In *Proceedings of the 2nd ISPRS International Workshop 3D-ARCH*, 2007, pp. 1–6.

[11] Baumberg, A. Blending Images for Texturing 3D Models. In *BMVC*, volume 3, Citeseer, 2002, p. 5.

[12] Callieri, M.; Cignoni, P.; et al. Masked photo blending: Mapping dense photographic data set on high-resolution sampled 3D models. *Computers & Graphics*, volume 32, no. 4, 2008: pp. 464–473.

[13] Kirschner, J. *Rekonstrukce 3D modelu z fotografií*. Master's thesis, České vysoké učení technické v Praze, 2008.

[14] Eisemann, M.; De Decker, B.; et al. Floating Textures. *Computer Graphics Forum (Proc. of Eurographics EG)*, volume 27, no. 2, Apr 2008: pp. 409–418, received the Best Student Paper Award at Eurographics 2008.

[15] Jeon, J.; Jung, Y.; et al. Texture map generation for 3D reconstructed scenes. *The Visual Computer*, volume 32, no. 6-8, 2016: pp. 955–965.

[16] Lempitsky, V.; Ivanov, D. Seamless mosaicing of image-based texture maps. In *Computer Vision and Pattern Recognition, 2007. CVPR'07. IEEE Conference on*, IEEE, 2007, pp. 1–6.

[17] Li, S. Z. Markov random field models in computer vision. In *European conference on computer vision*, Springer, 1994, pp. 361–370.

[18] Kindermann, R.; Snell, J. L. *Markov random fields and their applications*, volume 1. American Mathematical Society, 1980.

[19] Kato, Z.; Pong, T.-C. A Markov random field image segmentation model for color textured images. *Image and Vision Computing*, volume 24, no. 10, 2006: pp. 1103–1114.

[20] Besag, J. Spatial interaction and the statistical analysis of lattice systems. *Journal of the Royal Statistical Society. Series B (Methodological)*, 1974: pp. 192–236.

[21] Allène, C.; Pons, J.-P.; et al. Seamless image-based texture atlases using multi-band blending. In *Pattern Recognition, 2008. ICPR 2008. 19th International Conference on*, IEEE, 2008, pp. 1–4.

[22] Boykov, Y.; Veksler, O.; et al. Markov random fields with efficient approximations. In *Computer vision and pattern recognition, 1998. Proceedings. 1998 IEEE computer society conference on*, IEEE, 1998, pp. 648–655.

[23] Sobel, I. An isotropic 3× 3 image gradient operator. *Machine vision for three-dimensional scenes*, 1990: pp. 376–379.

[24] Simek, K. The Perspective Camera - An Interactive Tour. [online], 2012, [cit. 2018-01-09]. Available from: http://ksimek.github.io/2012/08/22/extrinsic/

[25] Gortler, S. J. *Foundations of 3D computer graphics*. MIT Press, 2012.

[26] WolfWings. Barrel distortion visual example. [online], september 2008, 2017-10-18 [cit. 2018-04-19]. Available from: https://commons.wikimedia.org/wiki/File:Barrel_distortion.svg

[27] Power, K. Backface culling. [online], 2012, [cit. 2018-01-09]. Available from: http://glasnost.itcarlow.ie/~powerk/GeneralGraphicsNotes/HSR/backfaceculling.html

[28] Kapoulkine, A. View frustum culling optimization – Representation matters. [online], 2009, [cit. 2018-02-22]. Available from: http://zeuxcg.blogspot.cz/2009/03/view-frustum-culling-optimization.html

[29] lighthouse3d. Geometric Approach – Testing Boxes. [online], 2015, [cit. 2018-02-22]. Available from: http://www.lighthouse3d.com/tutorials/view-frustum-culling/geometric-approach-testing-boxes/

[30] Marschner, S.; Shirley, P. *Fundamentals of computer graphics*. CRC Press, 2015.

[31] Bergen, G. v. d. Efficient collision detection of complex deformable models using AABB trees. *Journal of Graphics Tools*, volume 2, no. 4, 1997: pp. 1–13.

[32] Maini, R.; Aggarwal, H. Study and comparison of various image edge detection techniques. *International journal of image processing (IJIP)*, volume 3, no. 1, 2009: pp. 1–11.

[33] Fisher, R.; Perkins, S.; et al. Sobel Edge Detector. [online], 2004, [cit. 2018-02-30]. Available from: http://homepages.inf.ed.ac.uk/rbf/HIPR2/sobel.htm

[34] Thuerck, D.; Waechter, M.; et al. mapMAP MRF MAP Solver v1.3. [software], 2017-10-18 [cit. 2018-04-19]. Available from: https://github.com/dthuerck/mapmap_cpu

[35] Hecker, C. Perspective Texture Mapping. *Game Developer Magazine*, 1995.

[36] Thuerck, D.; Waechter, M.; et al. A Fast, Massively Parallel Solver for Large, Irregular Pairwise Markov Random Fields. In *Proceedings of High Performance Graphics 2016*, 2016.

[37] OpenCV team. OpenCV (Open Source Computer Vision Library). [software], 2018, version 3.4.1 [cit. 2017-10-25]. Available from: https://opencv.org

[38] G-Truc Creation. OpenGL Mathematics (GLM). [software], 2017, version 0.9.8.5 [cit. 2017-10-25]. Available from: https://glm.g-truc.net/0.9.8/index.html

[39] Microsoft Corporation. Configure an Ini File Item. [online], 2013, [cit. 2018-04-02]. Available from: https://docs.microsoft.com/en-us/previous-versions/windows/it-pro/windows-server-2008-R2-and-2008/cc731332(v=ws.11)

[40] Hoyt, B. inih (INI Not Invented Here). [software], 2018, version r41 [cit. 2018-02-05]. Available from: https://github.com/benhoyt/inih

[41] Ruhl, K. Coordinates: Bundler/VisualSFM, Matlab Calibration Toolbox, and OpenGL. [online], 2014, [cit. 2018-02-15]. Available from: http://www.land-of-kain.de/docs/coords/

[42] Cignoni, P.; Callieri, M.; et al. MeshLab: an Open-Source Mesh Processing Tool. In *Eurographics Italian Chapter Conference*, edited by V. Scarano; R. D. Chiara; U. Erra, The Eurographics Association, 2008, ISBN 978-3-905673-68-5, doi:10.2312/LocalChapterEvents/ItalChap/ItalianChapConf2008/129-136.

[43] Blender Online Community. Blender 2.78b. [software], 2017-10-18 [cit. 2018-04-19]. Available from: http://www.blender.org

[44] Samsung. Galaxy S8 S8+ specifications. [online], 2017, [cit. 2018-04-13]. Available from: http://www.samsung.com/global/galaxy/galaxy-s8/specs/

[45] Bourke, P. PLY-polygon file format. [online], 2009, [cit. 2018-04-22]. Available from: www:http://paulbourke.net/dataformats/ply

Appendix A

Acronyms

CTU Czech Technical University in Prague

FEE Faculty of Electrical Engineering

MRF Markov Random Field

MAP Maximum a posteriori

FOV Field of view

BVH Bounding volume hierarchy

AABB Axis aligned bounding boxes

UV Texture coordinates (u, v)

SFM Structure from motion

PLY Polygon File Format

Appendix B

Contents of enclosed media

- `README.md` ..... the readme file with project description
- `manual.pdf` ..... user manual
- `renders` ..... some texturing results
- `doc` ..... directory with Doxygen code documantation
- `TextureExtractorV2` ..... the directory of source codes
- `CMakeLists.txt` ..... main CMake file
- `example` ..... directory with demo example dataset
- `BP_Luzin_Danil_2018.pdf` ..... the thesis text in PDF format